Towards an Automated Approach to Use Expert Systems in the Performance Testing of Distributed Systems

A. Omar Portillo-Dominguez, Miao Wang, John Murphy
Lero, University College Dublin, Ireland
andres.portillo-dominguez@ucdconnect.ie; {miao.wang,j.murphy}@ucd.ie

Damien Magoni
LaBRI-CNRS, University of Bordeaux, France
magoni@labri.fr

Nick Mitchell,Peter F. Sweeney,Erik Altman
IBM T.J. Watson Research Center, New York, USA
{nickm,pfs,ealtman}@us.ibm.com

ABSTRACT

Performance testing in distributed environments is challenging. Specifically, the identification of performance issues and their root causes are time-consuming and complex tasks which heavily rely on expertise. To simplify these tasks, many researchers have been developing tools with built-in expertise. However limitations exist in these tools, such as managing huge volumes of distributed data, that prevent their efficient usage for performance testing of highly distributed environments. To address these limitations, this paper presents an adaptive framework to automate the usage of expert systems in performance testing. Our validation assessed the accuracy of the framework and the time savings that it brings to testers. The results proved the benefits of the framework by achieving a significant decrease in the time invested in performance analysis and testing.

Categories and Subject Descriptors

B.8 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Algorithms, Measurement, Performance

Keywords

Performance Testing, Automation, Performance Analysis, Expert Systems, Distributed Systems

1. INTRODUCTION

Performance is a critical dimension of quality, especially at enterprise-level, as it plays a central role in software usability. However it is not uncommon that performance issues materialise into serious problems (e.g., outages on production environments). For example, a 2007 survey applied to practitioners [5] reported that 50% of them had faced performance problems in at least 20% of their applications. Research studies have also documented the magnitude of this problem. For example, the authors of [7] found (in 2012) 332 previously unknown performance issues in the latest versions of five mature open-source software suites.

This situation is partially explained by the pervasive nature of performance, which makes it hard to assess as performance is influenced by every aspect of the design, code, and execution environment of an application. The latest trends in information technology (such as Cloud Computing[1]) have also augmented the complexity of applications, further complicating all activities related to performance. Under these conditions, it is not surprising that performance testing is complex and time-consuming. A special challenge is that performance tools heavily rely on human experts to be configured properly and to interpret their outputs [2, 14]. As this expertise is usually held by only a small number of people inside an organization, this issue could lead to bottlenecks, impacting the productivity of testing teams [2].

To simplify the performance analysis and testing, many researchers have been developing tools with built-in expertise [1, 2]. However, limitations exist in these tools that prevent their efficient usage in the performance testing of highly distributed environments. Firstly, these tools still need to be manually configured. If an inappropriate configuration is used, the tools might fail to obtain the desired outputs, resulting in significant time wasted. Secondly, testers need to manually carry out the data collections. In a distributed environment with multiple nodes to monitor and coordinate, such a manual process can be time-consuming and error-prone due to the vast amount of data to collect and consolidate. Similarly, excessive amounts of outputs produced by expert systems can overwhelm a tester due to the time required to correlate and analyse the results.

Even though these limitations might be manageable in small testing environments, they prevent the efficient usage of these tools in bigger environments. To exemplify this problem, consider the Eclipse Memory Analyzer Tool[2] (EMAT). If a tester wants to use EMAT in an environment composed of 100 nodes during a 24-hour test run and get results every hour, the tester would need to manually coordi-

[1]http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
[2]http://www.eclipse.org/mat/

nate the data gathering, the generation of the tool's reports, and the analysis of the reports. These steps conducted every hour, for a total of 2,400 iterations. As an alternative, the tester may focus the analysis on a single node, assuming it is representative of the whole system. However this assumption gives the risk of potentially overlooking issues.

An additional challenge is the overhead generated by any technique, which should be low to minimise the impact in the tested environment (e.g., inaccurate results or abnormal behaviour). Otherwise the technique might not be suitable for performance testing. Moreover, while manually using an expert system is not always applicable, its automation would encourage its usage. This strategy has already been proven successful in performance testing [6, 13].

This paper proposes an adaptive automation framework that addresses the common usage limitations of expert systems in performance testing. During our research we have successfully applied our framework to the IBM Whole-system Analysis of Idle Time tool (WAIT). This publicly available expert system helps to identify the performance inhibitors in Java systems. Our experimental results proved that the framework can configure WAIT without the need of manual tuning. The results also provided evidence about the benefits of the framework: The effort required to use and analyse the outputs of WAIT was reduced by 68%, while the total time spent on performance testing was reduced by 27%. The main contributions of this paper are:

1. A novel policy-based adaptive framework to automate the usage of expert systems in performance testing.
2. An adaptive policy to self-configure the gathering of data samples in an expert system, based on a desired performance threshold.
3. A practical validation of the framework and the policy, consisting of a prototype and two experiments. The first experiment proves the accuracy of the framework, and the second demonstrates the productivity benefits.

2. BACKGROUND AND RELATED WORK

Idle-time analysis is a methodology to identify the root causes of under-utilized resources. It is based on the behaviour that performance problems in multi-tier applications usually manifest as idle-time of waiting threads [1]. WAIT is an expert system that implements this methodology and has proven to simplify the detection of performance issues and their root causes in Java systems [1, 15]. WAIT is based on non-intrusive sampling mechanisms available at Operating System level (e.g., "ps" in Unix) and the Java Virtual Machine (JVM), in the form of *Javacores*[3] (snapshots of the JVM state, offering information such as threads). From an end-user perspective, WAIT is simple: A user only needs to collect as much data as desired, upload it to a public web page and get a report with the findings. This process can be repeated multiple times to monitor a system through time.

Given its strengths, WAIT is a promising candidate to reduce the dependence on a human expert and time required for performance analysis. However, as with other expert systems, the volume of data generated can be difficult to manage and the accuracy of WAIT depends on its configuration, where the preferable configuration might vary depending on the application and usage scenario. These characteristics make WAIT a good candidate to apply our framework.

[3]http://www-01.ibm.com/support/docview.wss?uid=swg27017906&aid=1

Automation in Testing. Most of the research has focused on automating the generation of load test suites. For example, the authors in [11] propose an approach to automate the generation of test cases based on specified levels of load and resources. Similarly, [4] presents an automation framework that separates the application logic from the performance testing scripts to increase the re-usability of the scripts. Meanwhile, [16] presents a framework designed to automate the performance testing of web applications.

Other research efforts have concentrated on automating specific analysis techniques. For example, [17] presents a combination of coverage analysis and debugging to automatically isolate failure-inducing changes. Similarly, the authors of [10] developed a technique to reduce the number of false memory leak warnings generated by static analysis techniques by automatically validating those warnings.

Finally, other researchers have proposed frameworks to support software engineering processes. For example, the authors of [3, 8] present frameworks to monitor software services. Both frameworks monitor the resource utilisation and the component interactions within a system. One focuses on Java [8] and the other on Microsoft technologies [3]. Unlike these works, which have been designed to assist on operational support activities, our framework is designed to address the specific needs of a tester in performance testing, isolating her from the complexities of an expert system.

3. ADAPTIVE FRAMEWORK

Our objective was to automate the manual processes involved in the usage of an expert system to improve a tester's productivity by decreasing the effort needed to use an expert system. Figure 1 depicts the contextual view of our framework, which executes concurrently with a performance test. It shields the tester from the complexities of the expert system, so that she only interacts with the load testing tool.

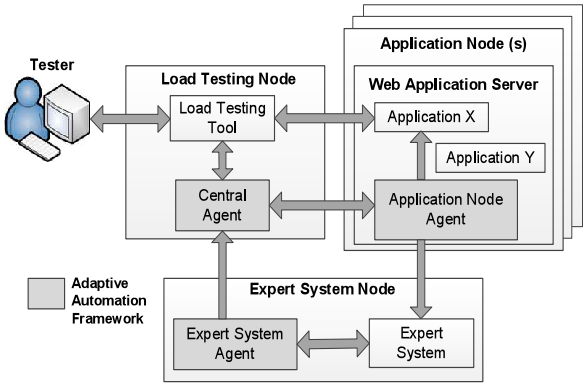

Figure 1: Proposed framework - Contextual view

Architecture. The proposed framework is implemented with the architecture depicted in Figure 1. It is composed of three types of agents: The *Central Agent* interacts with the load testing tool to know when the test starts and ends, evaluates the adaptive policies and propagates the decisions to the other nodes. The *Application Node Agent* performs any required tasks in each application node. Finally, the *Expert System Agent* interfaces with the expert system. All components communicate through commands, following the *Command*[4] Design Pattern.

[4]http://www.oodesign.com/command-pattern.html

Adaptive Automation Framework. Our framework is depicted in Figure 2. As a self-adaptive system is normally composed of a managed system and an autonomic manager [12], our framework plays the role of the autonomic manager. It controls the feedback loops which adapt the managed systems (the expert system and the application nodes under test) according to a set of goals.

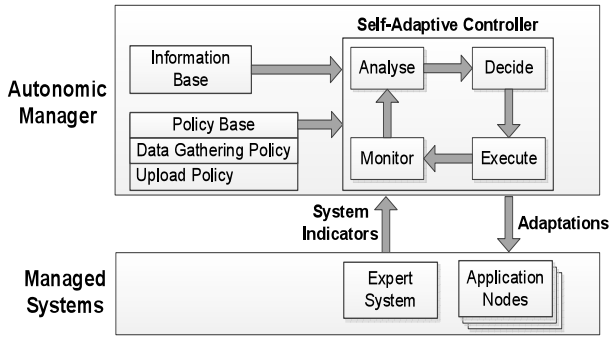

Figure 2: Adaptive automation framework

To incorporate self-adaptation in our framework, we follow the well-known MAPE-K adaptive model [9]. It is composed of 5 elements (depicted in Figure 2): A *Monitoring* element to obtain information from the managed systems; an *Analysis* element to evaluate if any adaptation is required; an element to *Plan* the adaptation, and an element to *Execute* it. The key element of our framework is its *policy base*, which fulfils the role of the *Knowledge* element and defines the pool of available adaptive policies. Each expert system must have at least two policies: A data gathering policy (to control the collection of samples), and an upload policy (to control when the samples are sent to the expert system for processing). An expert system might have other policies available (e.g., to back up the obtained samples).

From a configuration perspective, the tester needs to provide the *Information Base* (as shown in Figure 2), which is composed of all the inputs required by the chosen policies. For example, a data gathering policy might require a *Sampling Interval* to know the frequency for the collection of samples.

From a process perspective, the autonomic manager has a core process which coordinates the other MAPE-K elements. It is triggered when the performance test starts, and depicted in Figure 3. As an initial step, it gets a *Control Test Id*, value which uniquely identifies the test run and its collected data. This value is propagated to all the nodes. Next all application nodes start (in parallel) the loop specified in the monitor and analyse phases, until the test finishes: New data samples are collected following a data gathering policy. Then the analyser process checks the upload policies. If any has been fulfilled, the data is sent to the expert system (labelling the data with the *Control Test Id* so that information from different nodes can be identified as part of the same test). Similarly, updated results are retrieved from the expert system to be consolidated. Other policies might be executed depending on the user configuration. This core process continues iteratively until the performance test finishes. When that occurs, all applicable policies are evaluated one final time before the process ends.

Prototype. A prototype has been developed in conjunction with our industrial partner IBM. The *Central Agent* was

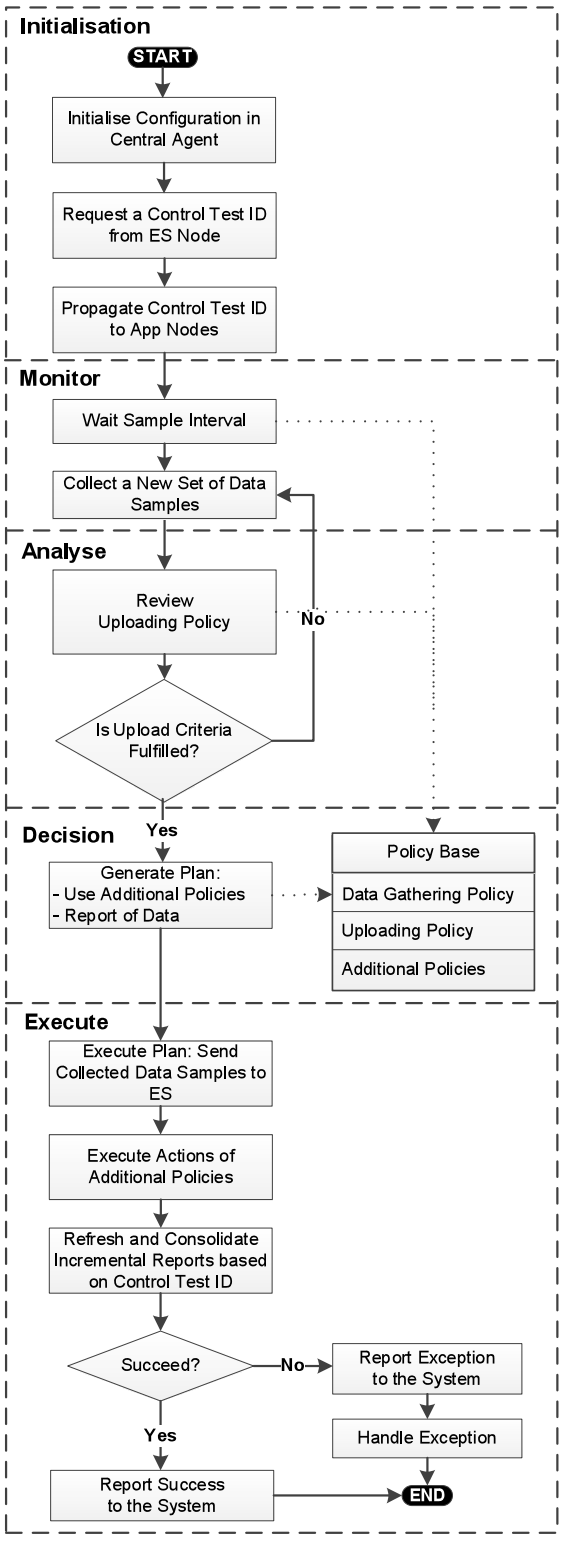

Figure 3: Automation framework - Core Process

implemented as a plugin for the Rational Performance Tester (RPT) [5], load testing tool commonly used in the industry and which eases the collection of testing metrics. The *Application Node Agent* was implemented as a Java Web Appli-

[5]http://www.ibm.com/software/products/en/performance

cation; and WAIT was the selected expert system due to its characteristics (discussed in Section 2). Finally, the *Expert System Agent* was implemented in PHP to extend the web interface of WAIT (which is developed in that technology). Two initial policies were also implemented: A data gathering policy with a constant *Sampling Interval* (SI) and an upload policy with a constant *Upload Time Threshold*(UTT).

4. ASSESSMENT OF TRADE-OFFS

To understand which adaptive policies would work best for WAIT, an assessment of its performance trade-offs was done. It involved comparing the throughput (TP) and the performance bugs identified by WAIT when using different configurations. These metrics were collected through RPT and the WAIT report, respectively.

All tests were done in an environment composed of eight VMs: Five application nodes, one WAIT server, one load balancer and one load tester (using RPT 8.5). All VMs had 2 virtual CPUs, 3GB of RAM, and 50GB of HD; ran Linux Ubuntu 12.04L, and Oracle Hotspot JVM 7 with a maximal heap of 2GB. The application nodes ran Apache Tomcat 6. The full DaCapo[6] benchmark 9.12 was chosen as application set because it offers a wide range of application behaviours to test. For each benchmark, its smallest *Sample Size* was used and each individual benchmark run was considered a transaction. To execute Dacapo from within a RPT HTTP test script, a wrapper JSP was developed. Also, a 24-hour test duration was used to reflect realistic test conditions.

As the SI controls the frequency of samples collection (main potential cause of overhead introduced by WAIT), a broad range of values was tested (0.125, 0.25, 0.5, 1, 2, 4, 8 and 16 minutes). The smallest value in the range (0.125 minutes) was chosen to be smaller than the minimum recommended value for WAIT (0.5 minutes). Similarly, the largest value was chosen to be larger than 8 minutes (a SI commonly used in the industry). As the UTT is not involved in the data gathering, a constant value of 30 minutes was used.

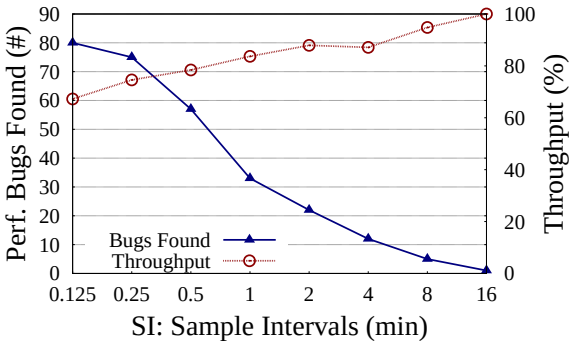

Figure 4: Perf. Bugs vs. Throughput

The results showed that there is a relationship between the selection of the SI and the performance cost of using WAIT. This behaviour is depicted in Figure 4, which summarizes the results of the tested configurations. It can be noticed how the throughput decreases when the SI decreases. This performance impact is mainly caused by the *Javacore* generation process which pauses the JVM during its execution[7]. Even though its cost was minimum with higher SIs, it gradually became visible (especially with SIs below 0.5 minutes). On the contrary, the number of identified bugs increases when the SI decreases. This positive impact is a direct consequence of feeding more samples to the WAIT server, which is pushed to do a more detailed analysis of the monitored application. A second round of analysis concentrated on the most critical issues identified by WAIT to assess if the previously described behaviours were also observed there (critical issues are defined as those ranked with an occurrence frequency above 80%). As shown in Figure 5, similar behaviours were confirmed. These observations led us to select the SI as an adaptive policy in our framework.

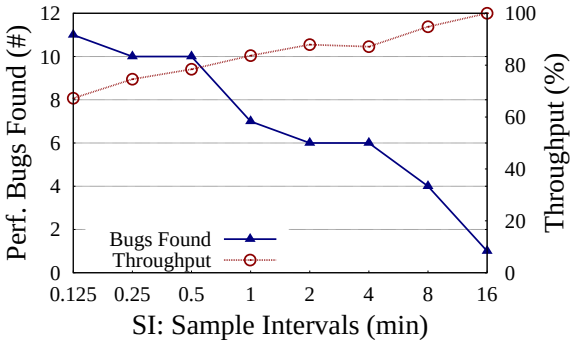

Figure 5: Perf. Critical Bugs vs. Throughput

5. ACCURACY-TARGET POLICY

This policy was designed to balance the trade-off between accuracy and performance, caused by the selection of the SI.

The policy process is depicted in Figure 6, where each step is mapped to the corresponding MAPE-K element. It requires two inputs: A response time threshold, which is the maximum acceptable impact to the response time (expressed as a percentage); and the warm-up period. Resembling its usage in performance testing[8], the warm-up period is the time after which all transactions have been executed at least once (hence contributing to the RT_{AVG} of the test run). Two other parameters are retrieved from our policy base, as their values are specific for each expert system: The minimum SI that should be used for collection; and the ΔSI, which indicates how much the SI should change in case of adjustment.

The process starts by waiting the configured warm-up period. Then it retrieves the average response time (RT_{AVG}) from the load testing tool. This value becomes the response time baseline (RT_{BL}). After that, the process initialises the application nodes with the minimum SI. This strategy allows collecting as many samples as possible, unless the performance degrades below the desired threshold, thus violating the SLA. Next, an iteratively monitoring process starts (which lasts until the test finishes): First, the process waits the current SI (as no performance impact caused by the expert system might occur until the data gathering occurs). Then, the new RT_{AVG} is retrieved and compared against the RT_{BL} to check if the threshold has been exceeded. If so, it means that the current SI is too small to keep the overhead below the configured threshold. In this case, the SI is increased by the value configured as ΔSI. Finally, the new SI is propagated to all the application nodes, which start using it since their next data gathering iteration.

[6]http://dacapobench.org/

[7]http://www-01.ibm.com/support/docview.wss?uid=swg27017906&aid=1

[8]http://msdn.microsoft.com/en-us/library/ff406976.aspx

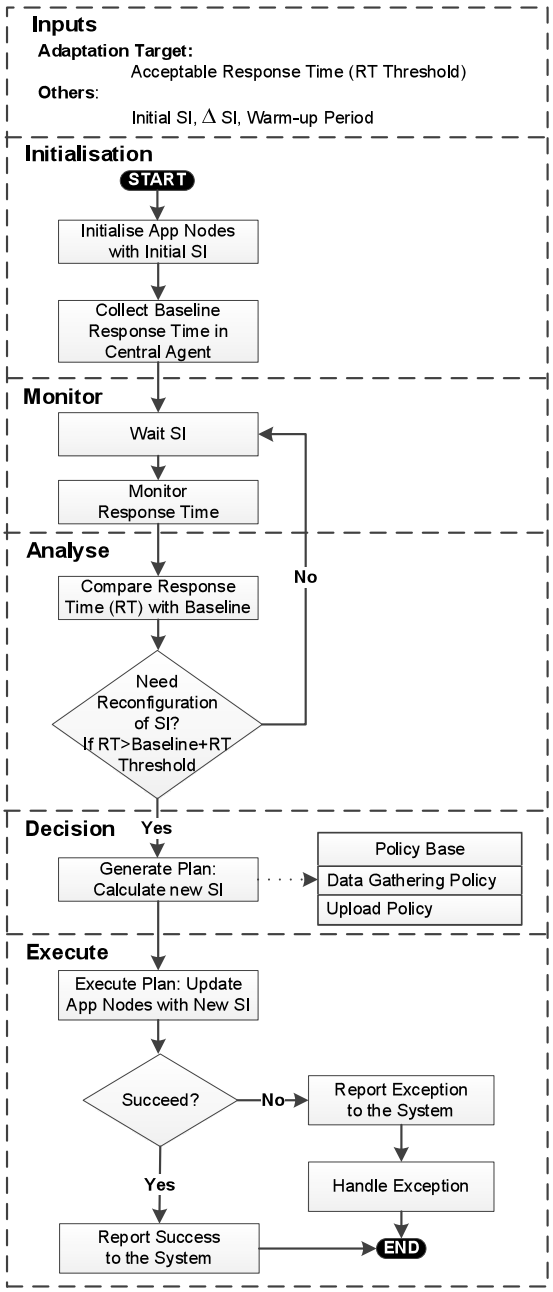

Figure 6: Accuracy-Target Data Gathering Policy

6. EXPERIMENTAL EVALUATION

Two experiments were done to evaluate the performance of our framework. The first evaluated the accuracy of the implemented policy, while the second assessed the productivity gains of our framework. The next sections describe these experiments and their results.

Assessment of Adaptive Policy. The objective was to evaluate if the adaptive policy fulfilled its purpose of configuring the tool without the need for manual intervention. The set-up was equal to the one used in the assessment of the WAIT trade-offs (Section 4) except the SI selection, as the adaptive policy took the place of this manual configuration.

The adaptive policy used a 20% response time threshold, (value suggested by IBM to reflect real-world conditions); and a warm-up period of 5 minutes (found to be enough for all test transactions to execute at least one). Finally, the minimum SI and the ΔSI were set to 30 seconds.

The results obtained were compared against the results from the assessment of WAIT trade-offs (Section 4). This demonstrated that the accuracy policy worked, as it was possible to finish the test with the overhead caused by WAIT within the desired threshold. This was the result of increasing the SI when the threshold was exceeded to reduce the performance impact. In our case, this adjustment involved that the SI was increased twice, moving from its initial value of 30 seconds to 60 seconds, then to a final value of 90 seconds. Also, the number of bugs found with the adaptive policy was higher than those found with all the static SIs which fulfilled the response time threshold of 20% (the SIs of 1 minute or higher). This was the result of using other (smaller) SIs during the test, which yielded a better bug coverage during certain periods of the test. This behaviour also proved that the policy could avoid the need of manually tuning the tool, as it automatically identified the best SI configuration (which varied during the test). The same analysis was done considering only the critical bugs, and similar behaviours were observed. These results are presented in Figures 7 (overall bugs) and 8 (critical bugs), where the response time threshold is shown as a grey horizontal line.

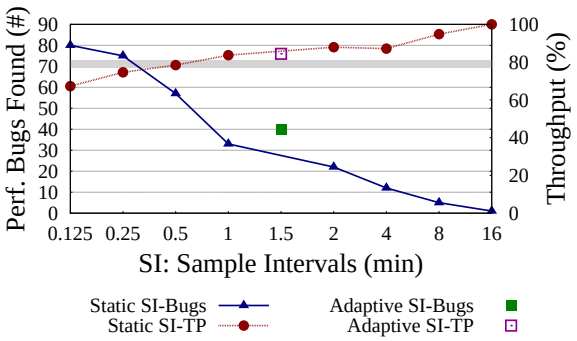

Figure 7: Perf. Bugs vs. Throughput

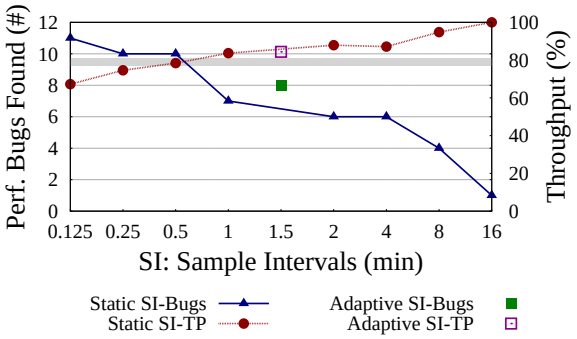

Figure 8: Critical Perf. Bugs vs. Throughput

Assessment of benefits in testing productivity. The objective was to assess the benefits that our framework brings to a tester in terms of reduced effort and time. The set-up was similar to the one used in the previous experiment with two exceptions: First, the well-documented open source iBatis JPetStore 4.0 [9] application was used with a workload of 2,000 concurrent users. Second, the number of application nodes was increased (to 10 nodes) to test our framework

[9]http://sourceforge.net/projects/ibatisjpetstore/

in a bigger test environment. This experiment also involved modifying the source code of JPetStore to manually inject five common performance issues (two lock contentions, two deadlocks and one I/O latency bug).

Two types of runs were performed: The first type involved a tester using WAIT manually (M-WAIT). A second type of run involved using WAIT through our automation framework (A-WAIT). In both cases, the tester did not know the number or characteristics of the injected bugs.

The results are summarized in Table 1. After comparing the runs, two time savings were documented when using A-WAIT: First, the effort to identify bugs was decreased 68%. This was the result of simplifying the analysis of the WAIT reports: Instead of having multiple reports to be manually correlated, the tester using A-WAIT only needed to monitor a single report which incrementally evolved. This allowed the tester using A-WAIT to get intermediate results during the test run. In our case, all bugs were identified after the first hour of test execution. Thus, she was able to start the analysis of the bugs in parallel to the rest of the test execution (which she kept monitoring). A consequence of this was that the duration of the performance testing activity decreased 27%. It is worth mentioning that both testers were able to identify all bugs with the help of the WAIT reports.

Table 1: M-WAIT and A-WAIT Comparison

Metric	M-WAIT (hr)	A-WAIT (hr)	M-WAIT vs. A-WAIT (%)
a. Duration of Perf. testing activity	32.8	24.1	-27%
b. Duration of Perf. Testing	24.0	24.0	0%
c. Effort of Perf. Analysis (d+e)	8.8	4.2	-52%
d. Effort of Bug Identification	6.8	2.2	-68%
e. Effort of Root Cause Analysis	2.0	2.0	0%

An additional observation from this experiment is that the time savings gained by the automated framework are directly related to the duration of the test and the number of application nodes in the environment. This behaviour is especially valuable in long-term runs, which are common in performance testing and typically last several days. The same situation occurs with the performance testing of highly distributed environments, as the obtained time savings will be higher under those conditions.

To summarize the experimental results, they allowed the measurement of the productivity benefits that a tester can gain by using an expert system through our proposed automation framework. A direct consequence of these time savings is the reduction of expert knowledge dependency and effort required to identify performance issues, hence improving the productivity of testers.

7. CONCLUSIONS AND FUTURE WORK

The identification of performance problems in highly distributed environments is complex. Even though researchers have been developing expert systems to simplify this task, limitations still exist in such systems that prevent their effective usage in performance testing. To address these limitations, this paper proposed an adaptive policy-enabled framework to automate the usage of an expert system in a distributed testing environment. In particular, our experimental results showed significant time savings gained by applying the proposed framework: The effort required to identify bugs was reduced by 68%(compared to the manual usage of the expert system), while the total duration of the performance testing activity was reduced by 27%. The results also proved that such an adaptive framework is capable of simplifying the usage of an expert system. This was achieved by balancing the effectiveness of the tool and its overhead without manual configuration. Future work will focus on extending the adaptive capabilities of our framework.

8. ACKNOWLEDGMENTS

Supported, in part, by Science Foundation Ireland grant 10/CE/I1855. Our thanks also to Patrick O'Sullivan and Amarendra Darisa, as their experience in performance testing helped us through the scope definition and validation.

9. REFERENCES

[1] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. *ACM SIGPLAN Notices*, Oct. 2010.

[2] V. Angelopoulos, T. Parsons, J. Murphy, and P. O'Sullivan. GcLite: An Expert Tool for Analyzing Garbage Collection Behavior. *ACSACW*, 2012.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. *OSDI*, 2004.

[4] S. Chen, D. Moreland, S. Nepal, and J. Zic. Yet Another Performance Testing Framework. *Australian Conference on Software Engineering*, 2008.

[5] Compuware. *Applied Performance Management Survey*. 2007.

[6] S. Dosinger, Stefan, Richard Mordinyi. Communicating CI servers for increasing effectiveness of automated testing. *ASE*, 2012.

[7] E. G. Jin, L. Song. Understanding and detecting real-world performance bugs. *PLDI*, 2012.

[8] V. Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. 2009.

[9] D. J. Kephart. The vision of autonomic computing. *Computer*, Jan. 2003.

[10] E. Li, Mengchen. Dynamically Validating Static Memory Leak Warnings. *ISSTA*, 2013.

[11] J. M. S. Bayan. Automatic stress and load testing for embedded systems. *ICSA*, 2006.

[12] L. M. Salehie. Self-adaptive software: Landscape and research challenges. *TAAS*, 2009.

[13] S. Shahamiri, W. Kadir, and S. Mohd-Hashim. A Comparative Study on Automated Software Test Oracle Methods. ICSEA, 2009.

[14] M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. *FOSE*, 2007.

[15] T. Wu, Haishan, Asser N. Tantawi. A Self-Optimizing Workload Management Solution for Cloud Applications. 2012.

[16] W. Xingen Wang, Bo Zhou. Model-based load testing of web applications. *Journal of the Chinese Institute of Engineers*, 2013.

[17] E. Yu, Kai. Practical isolation of failure-inducing changes for debugging regression faults. *ASE*, 2012.